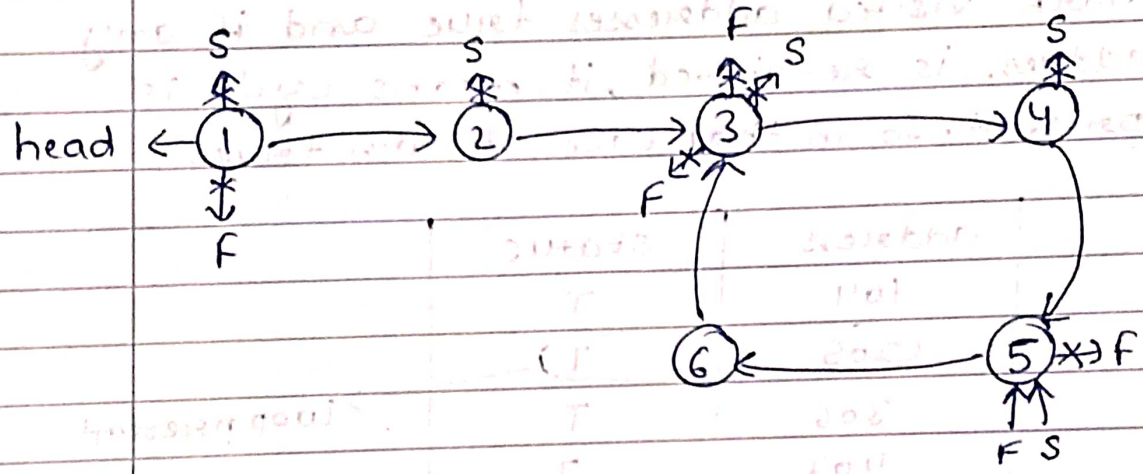
Note - Jb bhi koi ques ko map se solve kr rhe honge. We should have an alternative approach for that question even if that solution takes more time & space complexity than map solution.

06/11/2023
Monday

* Linked list cycle (leetcode -141)

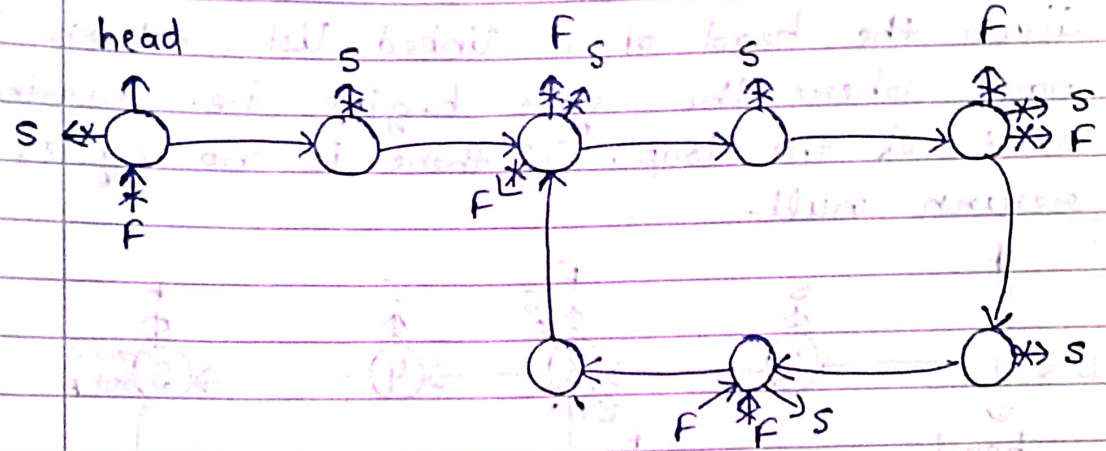
Using slow and faster pointer approach



Approach -

(1) Set Fast and slow pointers at head.
(2) Fast → 2 step , slow → 1step
(3) If loop is present ; then fast and slow pointers meet at a node. This is the node which tells that linked list has cycle in it.

Why this approach is working?



When both slow and fast pointers enters in the loop, distance b/w slow & fast pointers is 3, 2, 1, 0 nodes and at one node, they will meet.

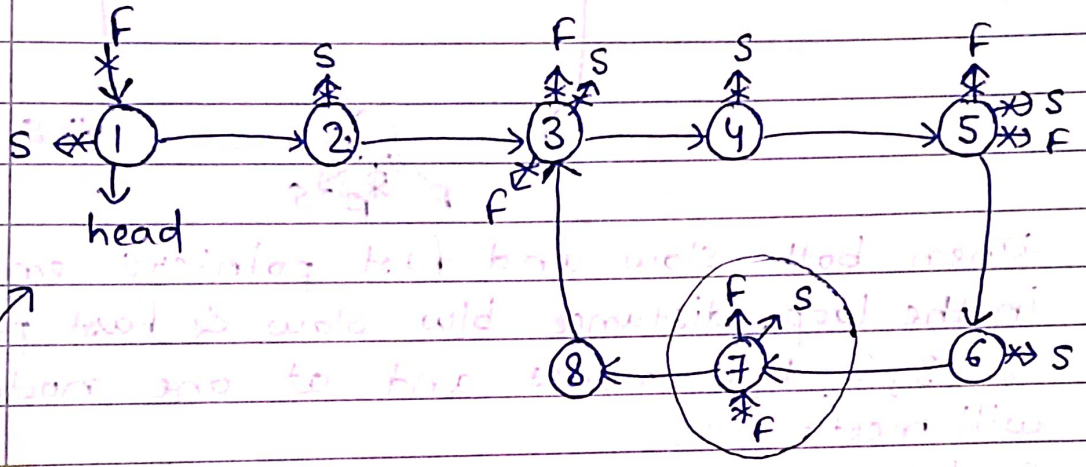
So, here we can visualize that the distance b/w the slow and fast pointers is decreasing eventually and at one point, they will must meet if there is a loop in LL.

Code-

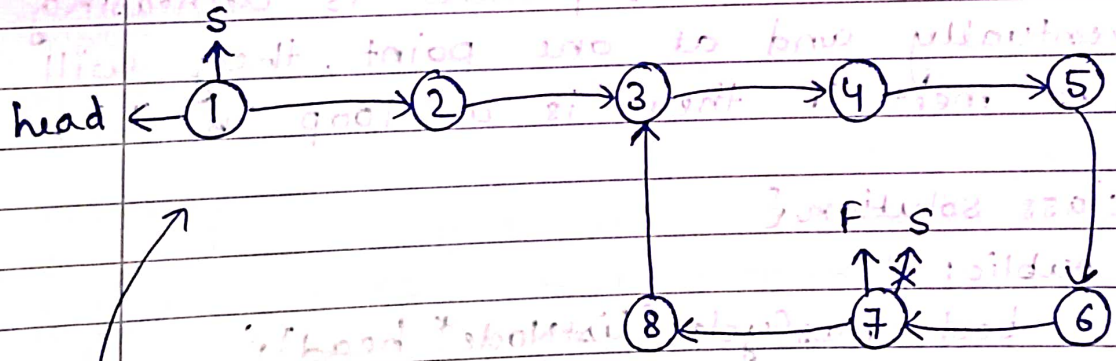
```
class solution{
  public:
    bool hasCycle (ListNode* head){
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast != NULL){
            fast = fast->next;
          if (fast != NULL){
            fast = fast->next;
            slow = slow->next;
        }
      if(fast == slow){
          return true; }
      } return false; } };
```

* Linked list cycle -II (leetcode - 142)

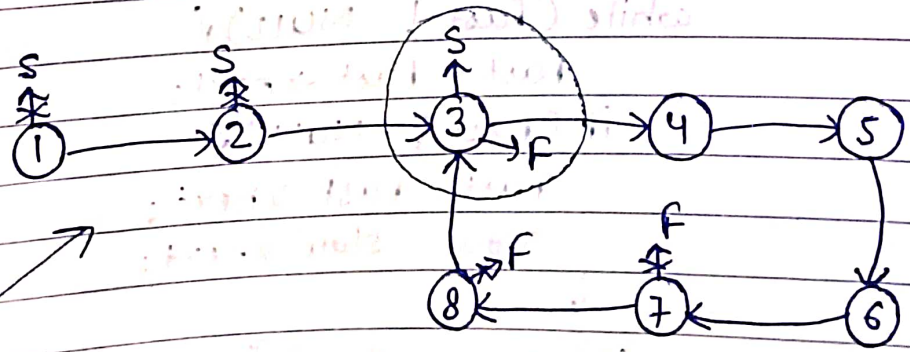
Given the head of a linked list, return the node where the cycle begins i.e. starting point of the loop. If there is no cycle, return null.



Step 1 - check if cycle is present.



Step 2 - Set slow at head.



step 3 - Increment fast & slow by 1 step. The point where they meet is starting point of the loop.

Algorithm –

Step 1 – Check cycle in linked list.

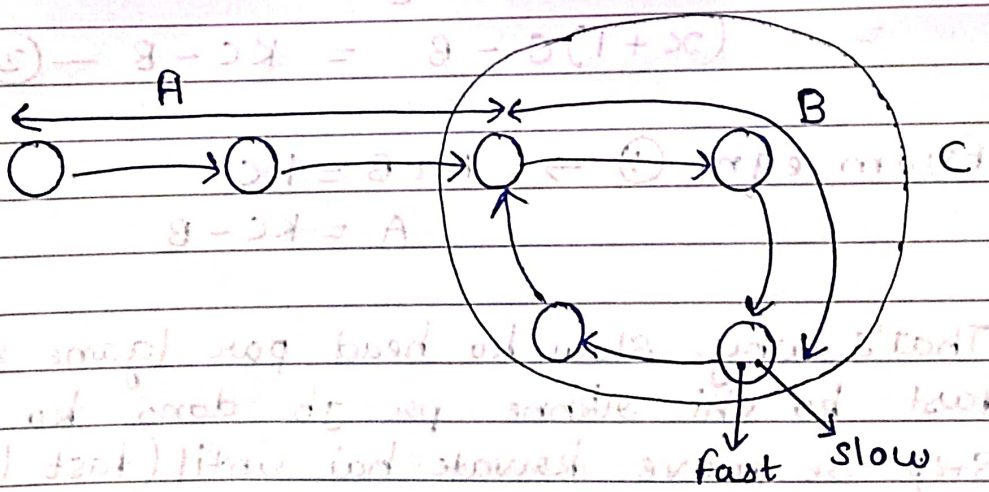
Step 2 – fast == slow

loop present → (A) slow = head.

(B) slow, fast → 1 step increment until they meet.

(C) Return either fast or slow.

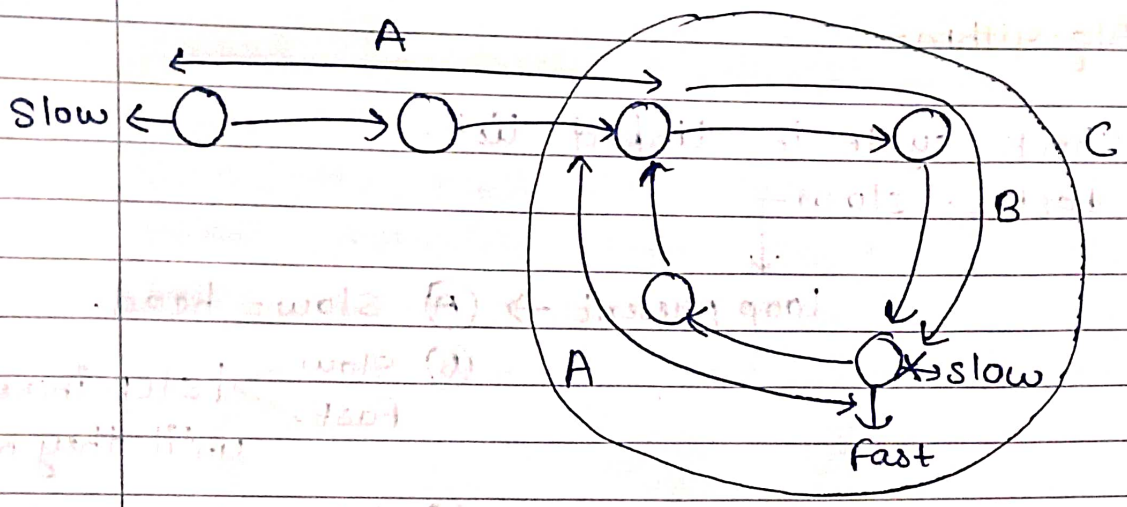
Why this algorithm is working ?



As, distance travelled by fast pointer = 2 x distance travelled by slow pointer

i.e. $A + \text{some cycles} + B = 2(A + \text{some other cycles} + B)$

$$A + K_1C + B = 2(A + K_2C + B)$$
$$A + K_1C + B = 2A + 2K_2C + 2B$$
$$K_1C - 2K_2C = 2A + 2B - A - B$$
$$K \leftarrow (K_1 - 2K_2)C = A + B \quad \text{i.e.} \quad \boxed{A + B = KC} \quad -(1)$$



Distance travelled by fast after the above positions of fast and slow –

$$= \text{some cycles} + C - B$$

$$= x\,C + C - B$$

$$= (x+1)C - B \quad = KC - B \quad -(2)$$

From eqn. (1) → $A + B = KC$

$$A = KC - B$$

That's why slow ko head par lgane se and fast ko vhi rkhne pr jb dono ko 1-1 step se move krvate hai until (fast != slow) dono pointers starting point of the loop pr meet kar jate hai.

Code -

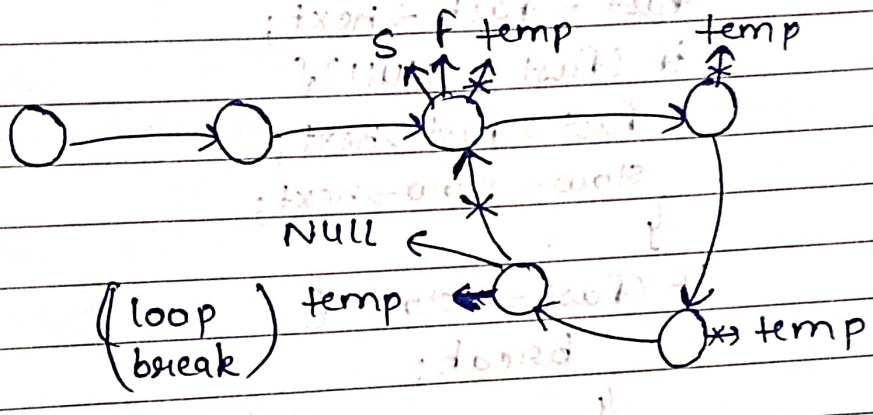
```
class solution{
    public:
ListNode * detectCycle (ListNode* head) {
    ListNode* slow =head;
    ListNode * fast = head;

    while ( fast != NULL){
        fast = fast ->next;
        if (fast != NULL){
            fast = fast ->next;
            slow = slow->next;
        }
        if (fast == slow){
            break;
        }
    }
if (fast == NULL){
    //cycle is absent
    return NULL;
}

    slow= head;
    while (fast != slow){
        fast = fast ->next;
        slow = slow ->next;
    }
    return fast;
}
};
```

* **Remove loop in linked list.-**

Step 1- Find starting point of the loop.

Step 2- Make a temp pointer that starts from starting point of loop and temp = temp -> next until temp -> next equals starting point of loop.

Step 3- temp -> next = NULL.



Code-

```
class solution {
    public:
void removeloop (Node * head) {
  Node * slow = head;
  Node * fast = head;
  while (fast != NULL) {
      fast = fast -> next;
    if (fast != NULL){
      fast = fast -> next;
      slow = slow -> next;
    }
  if (fast == slow) {
      break;
  }
}
if (fast == NULL) {
    return;
}
```

```
slow = head;
while (fast != slow){
    fast= fast ->next;
    slow= slow ->next;
  }
Node * startingPoint = fast;
Node * temp = startingPoint;
while (temp ->next != startingPoint) {
        temp = temp ->next;
    }
    temp ->next = NULL;
  }
};
```

* **Add one to linked list -**

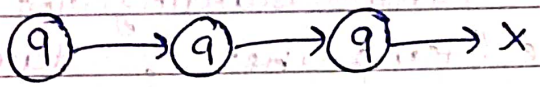
Step 1 - reverse.
Step 2 - Add one.
Step 3 - reverse.

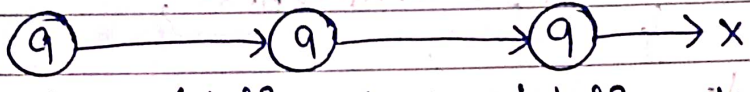
2 observations →
- if (carry==0), in that case break kr jao.
- if (carry != 0) and last node pr temp stand kar rha h, in that case new node create krni pdegi or usme carry ko push krna hai.

Note - LL ko reverse krne ki need isliye pdi coz addition right to left hota hai but LL mai right to left jana possible nhi hai. That's why reverse krke fir add kiya.

__/__/____

Note - LL ki last node ko alg se process krna pdega kyuki chances hai carry not equal 0 ho and LL khtm ho jaye.. So, new node add krni pdegi. That's why temp->next != null krenge.

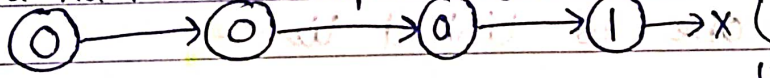
for example - I/P = 

After reversing- 

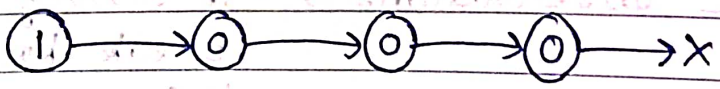
totalsum=10	totalSum=10	totalSum=10
digit=10%10=0	digit=10%10=0	digit=10%10=0
carry=10/10=1	carry=10/10=1	carry=10/10=1
temp->data=digit	temp->data=digit	temp->data=digit

ye wala kaam while loop ne kiya

Is node ko alg process kiya tha, that's why new node add kiya or usme carry push kr diya.



(carry != 0)

Reversing again- 

Code -

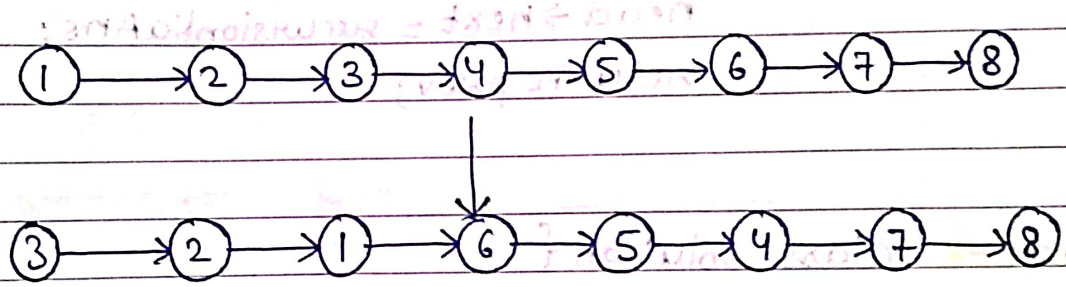
```
Node* reverseLL (Node* &head){
    Node* prev = NULL;
    Node* curr = head;
    while (curr != NULL){
        Node * nextNode = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextNode;
    }
    return prev;
}

void addOne (Node* head){

    //reverse
    head = reverseLL (head);

    // add one
    Node * temp = head;
    int carry = 1;
    while (temp->next != NULL){
        int totalSum = temp->data + carry;
        int digit = totalSum % 10;
        carry = totalSum / 10;

        temp->data = digit;
        temp = temp->next;
        if (carry == 0){
            break;
        }
    }
```

```
// processings for the last node
if (carry != 0) {
    int totalSum = temp->data + carry;
    int digit = totalSum % 10;
    carry = totalSum / 10;

    temp->data = digit;
    if (carry != 0){
        Node * newNode = new Node (carry);
        temp->next = newNode;
    }
}
// reverse
head = reverseLL (head);
}
```

* **Reverse nodes in k-Group (leetcode-25)**

Given the head of a linked list, reverse the nodes of the list k at a time and return the modified list.
k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k, then left out nodes, in the end, should remain as it is.
You may not alter the values in list's nodes, only nodes themselves may be changed.

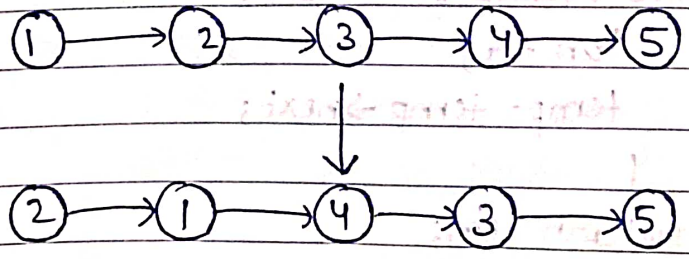
Example1-



I/P - head = [1,2,3,4,5,6,7,8], k=3

O/P - [3,2,1,6,5,4,7,8]

Example 2 -

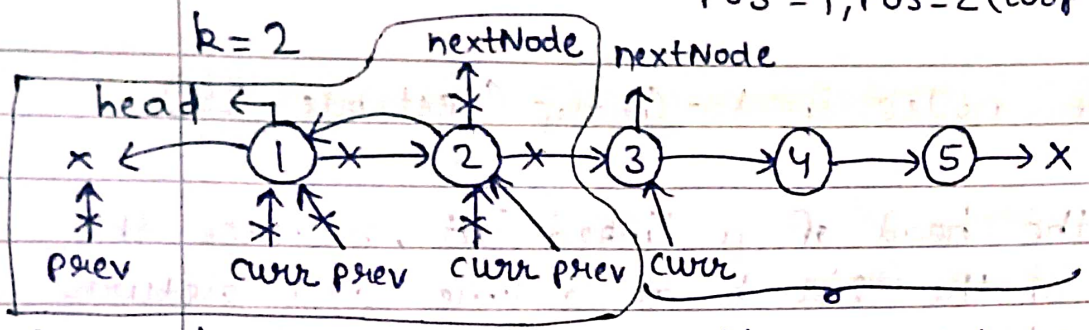


I/P - head= [1,2,3,4,5], k=2

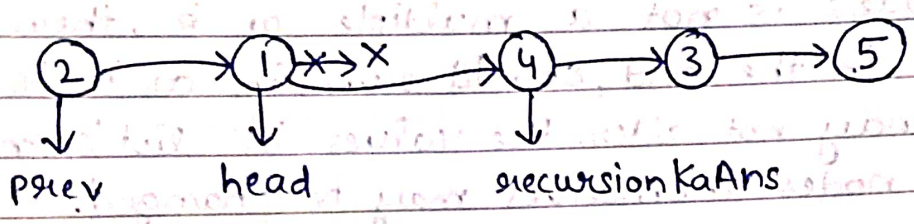
O/P - [2,1,4,3,5]

Pos = 0

Pos = 1, Pos = 2 (loop break)



1 case humne solve kiya Ye rest part recursion krega




```
head->next = recursionKaAns;
return prev;
```

Code -

```
class solution{
public:

int getlength (ListNode* head){
    ListNode *temp = head;
    int len=0;
    while (temp != NULL){
        len ++;
        temp= temp->next;
    }
    return len;
}
```

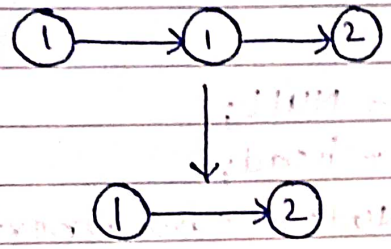
```
ListNode * reverseKGroup (ListNode* head, int k){

    // base case
    if (head == NULL || head->next == NULL){
        return head;
    }

    ListNode * prev = NULL;
    ListNode * curr = head;
    ListNode * nextNode = curr->next;
    int pos = 0;
    int len = getlength (head);
    if (len < k){
        return head;
    }
    while (pos < k){
        pos++;
        nextNode = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextNode;
    }
    ListNode * recAns = NULL;
    if (curr != NULL){
        recAns = reverseKGroup (nextNode, k);
        head->next = recAns;
    }
    return prev;
}
};
```

// new head
// is curr or
// nextNode
// Both are at
// same position

* ; **Remove duplicates from sorted list (leetcode - 83)**

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

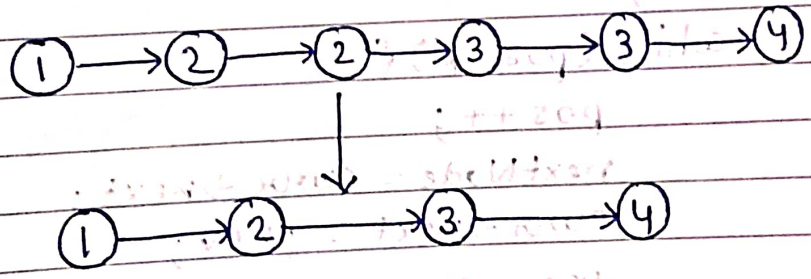
Example 1 -



I/P - head = [1, 1, 2]

O/P - [1, 2]

Example 2 -

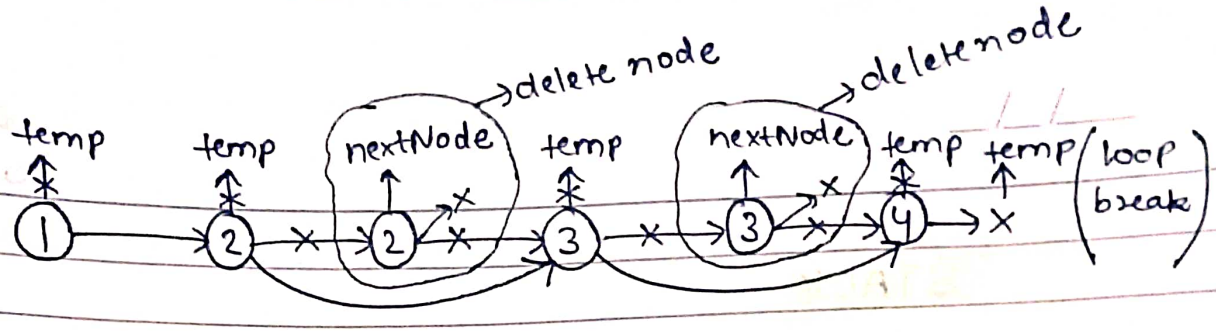


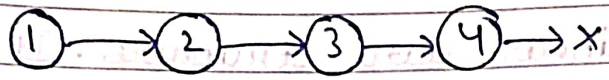
I/P - head = [1, 2, 2, 3, 3, 4]

O/P - [1, 2, 3, 4]

Logic -

```
while (temp != NULL){
  if (temp->next != NULL && temp->val == temp->next->val)
    {
      //remove
    }
  else {
      temp = temp->next;
    }
}
```





Code-

```
class solution{
public:
    listNode* deleteDuplicates(listNode* head){
        if(head==NULL || head->next==NULL){
            return head;
        }
    listNode *temp=head;
    while(temp!=NULL){
        if(temp->next!=NULL && temp->val==
                                temp->next->val)
        {
        listNode *nextNode = temp->next;
        temp->next = nextNode->next;
        nextNode->next = NULL;
        delete nextNode;
        }
        else{
            temp= temp->next;
        }
    }
    return head;
  }
};
```